# Networking IPv6 User Guide for JDK/JRE 5.0

This document covers the following topics:


- Overview
- Supported Operating Systems
- Using IPv6 in Java
- Details on IPv6 Support in Java
  - Special IPv6 Address Types
  - IPv6-Related System Properties
  - Dual-Stack Node
  - Java Application Impact
  - IPv6 Networking Properties


## Overview

Within the past few years IPv6 has gained much greater acceptance in the industry, especially in certain regions of the world, i.e., Europe and the Asian Pacific. Extensibility, mobility, quality of service, larger address space, auto-configuration, security, multi-homing, anycast and multicast, and renumbering—these are some of the features of IPv6 that make it desirable.

With the release of J2SE 1.4 in February 2002, Java began supporting IPv6 on Solaris and Linux. Support for IPv6 on Windows was added with J2SE 1.5. While other languages, such as C and C++ can support IPv6, there are some major advantages to Java:

- With Java you invest in a single code base that is both IPv4- and IPv6-ready.
- Your existing Java applications are already IPv6-enabled.
- Migration to IPv6 is easy

We will prove these statements with code examples below and provide additional details on IPv6 support.

## Supported Operating Systems

The following operating systems are now supported by the J2SE reference implementation:

- Solaris 8 and higher
- Linux kernel 2.1.2 and higher (kernel 2.4.0 and higher recommended for better IPv6 support)
- Windows XP SP1 and Windows 2003

## Using IPv6 in Java

Using IPv6 in Java is easy; it is transparent and automatic. Unlike in many other languages, no porting is necessary. In fact, there is no need to even recompile the source files.

Consider an example from The Java Tutorial:

```
Socket echoSocket = null;
   PrintWriter out = null;
   BufferedReader in = null;

   try {
      echoSocket = new Socket("taranis", 7);
      out = new PrintWriter(echoSocket.getOutputStream(), true);
      in = new BufferedReader(new InputStreamReader(
         echoSocket.getInputStream()));
   } catch (UnknownHostException e) {
      System.err.println("Don't know about host: taranis.");
      System.exit(1);
   } catch (IOException e) {
      System.err.println("Couldn't get I/O for "
         + "the connection to: taranis.");
      System.exit(1);
   }
// ... code omitted here
communicateWithEchoServer(out, in);

out.close();
```

```
in.close();
stdIn.close();
echoSocket.close();
```

You can run the same bytecode for this example in IPv6 mode if both your local host machine and the destination machine (taranis) are IPv6-enabled.

In contrast, if you wanted the corresponding C program to run in IPv6 mode, you would first need to port it. Here's what would need to happen:

**Excerpt of original C code:**

```
struct sockaddr_in sin;
struct hostent *hp;
int sock;

/* Open socket. */
sock = socket(AF_INET, SOCK_STREAM, 0);
if (sock == -1) {
   perror("socket");
   return (-1);
}

/* Get host address */
hp = gethostbyname(hostname);
if (hp == NULL || hp->h_addrtype != AF_INET || hp->h_length != 4) {
   (void) fprintf(stderr, "Unknown host '%s'\n", hostname);
   (void) close(sock);
   return (-1);
}

sin.sin_family = AF_INET;
sin.sin_port = htons(port);
(void) memcpy((void *) &sin.sin_addr, (void *)hp->h_addr, hp->h_length);

/* Connect to the host */
if (connect(sock, (struct sockaddr *)&sin, sizeof(sin)) == -1) {
   perror("connect");
   (void) close(sock);
   return (-1);
}
```

**Modified IPv6-aware C code:**

```
struct addrinfo *res, *aip;
struct addrinfo hints;
int sock = -1;
int error;

/* Get host address. Any type of address will do. */
bzero(&hints, sizeof(hints));
hints.ai_flags = AI_ALL|AI_ADDRCONFIG;
hints.ai_socktype = SOCK_STREAM;

error = getaddrinfo(hostname, servicename, &hints, &res);
if (error != 0) {
   (void) fprintf(stderr,
      "getaddrinfo: %s for host %s service %s\n",
      gai_strerror(error), hostname, servicename);
   return (-1);
}
/* Try all returned addresses until one works */
for (aip = res; aip != NULL; aip = aip->ai_next) {
   /*
   * Open socket. The address type depends on what
   * getaddrinfo() gave us.
   */
   sock = socket(aip->ai_family, aip->ai_socktype, aip->ai_protocol);
   if (sock == -1) {
      perror("socket");
      freeaddrinfo(res);
      return (-1);
   }

/* Connect to the host. */
if (connect(sock, aip->ai_addr, aip->ai_addrlen) == -1) {
   perror("connect");
   (void) close(sock);
```

```
        sock = -1;
        continue;
    }
    break;
    }
        freeaddrinfo(res);
```

Note that for new applications, if you write address-family-agnostic data structures, there is no need for porting.

However, when it comes to server-side programming in C/C++, there is an additional wrinkle. Namely, depending on whether your application is written for a dual-stack platform, such as Solaris or Linux, or a single-stack platform, such as Windows, you would need to structure the code differently. For server-side programming, Java shows a big advantage. You still write the same code as before:

```
ServerSocket server = new ServerSocket(port);
Socket s;
while (true) {
   s = server.accept();
   doClientStuff(s);
}
```

Now, however, if you run the code on an IPv6-enabled machine, you immediately have an IPv6-enabled service.

Here's the corresponding server C code for a dual-stack platform:

```
int ServSock, csock;
struct sockaddr addr, from;
...
ServSock = socket(AF_INET6, SOCK_STREAM, PF_INET6);
bind(ServSock, &addr, sizeof(addr));
do {
   csock = accept(ServSocket, &from, sizeof(from));
   doClientStuff(csock);
} while (!finished);
```

Notice that on a dual-stack machine, since one socket, the IPv6 socket, will be able to access both IPv4 and IPv6 protocol stacks, you only need to create one socket. Thus this server can potentially support both IPv4 and IPv6 clients.

Here's the C code for the same server for a single-stack platform:

```
SOCKET ServSock[FD_SETSIZE];
ADDRINFO AI0, AI1;
ServSock[0] = socket(AF_INET6, SOCK_STREAM, PF_INET6);
ServSock[1] = socket(AF_INET, SOCK_STREAM, PF_INET);
 ...
bind(ServSock[0], AI0->ai_addr, AI0->ai_addrlen);
bind(ServSock[1], AI1->ai_addr, AI1->ai_addrlen);
 ...
select(2, &SockSet, 0, 0, 0);
if (FD_ISSET(ServSocket[0], &SockSet)) {
   // IPv6 connection csock = accept(ServSocket[0], (LPSOCKADDR)&From, FromLen);
    ...
}
if (FD_ISSET(ServSocket[1], &SockSet))
   // IPv4 connection csock = accept(ServSocket[1], (LPSOCKADDR)&From, FromLen);
    ...
}
```

Here you need to create two server sockets, one for IPv6 stack and one for IPv4 stack. You also need to multiplex on the two sockets to listen to connections from either IPv4 or IPv6 clients.

With Java you can run any Java applications, client or server, on an IPv6-enabled platform using J2SE 1.4 or later, and that application will automagically become IPv6-enabled.

Contrasting this with legacy, native-language applications, if you wanted any C/C++ applications to be IPv6-enabled, you would need to port and recompile them.

**How IPv6 Works on a Java Platform**

The Java networking stack will first check whether IPv6 is supported on the underlying OS. If IPv6 is supported, it will try to use the IPv6 stack. More specifically, on dual-stack systems it will create an IPv6 socket. On separate-stack systems things are much more complicated. Java will create two sockets, one for IPv4 and one for IPv6 communication. For client-side TCP applications, once the socket is connected, the internet-protocol family type will be fixed, and the extra socket can be closed. For server-side TCP applications, since there is no way to tell from which IP family type the next client request will come, two sockets need to be maintained. For UDP applications, both sockets will be needed for the lifetime of the communication.

Java gets the IP address from a name service.

# Details on IPv6 support in Java

You don't need to know the following in order to use IPv6 in Java. But if you are curious and what to know what happens under various circumstances, the remainder of this document should provide answers.

## Special IPv6 Address Types

### Unspecified address (:: corresponding to 0.0.0.0 in IPv4)

This is also called *anylocal* or *wildcard* address. If a socket is bound to an IPv6 anylocal address on a dual-stack machine, it can accept both IPv6 and IPv4 traffic; if it is bound to an IPv4 (IPv4-mapped) anylocal address, it can only accept IPv4 traffic. We always try to bind to IPv6 anylocal address on a dual-stack machine unless a related system property is set to use IPv4 Stack.

When bound to : :, method `ServerSocket.accept` will accept connections from both IPv6 or IPv4 hosts. The Java platform API currently has no way to specify to accept connections only from IPv6 hosts.

Applications can enumerate the interfaces using `NetworkInterface` and bind a `ServerSocketChannel` to each IPv6 address, and then use a selector from the New I/O API to accept connections from these sockets.

> **Note:** The option discussed below is introduced in Draft-ietf-ipngwg-rfc2553bis-03.txt. It will be supported in the Java 2 platform when it becomes a standard.

However, there is a new socket option that changes the above behaviour. Draft-ietf-ipngwg-rfc2553bis-03.txt has introduced a new IP level socket option, IPV6_V6ONLY. This socket option restricts AF_INET6 sockets to IPv6 communications only. Normally, AF_INET6 sockets may be used for both IPv4 and IPv6 communications. Some applications may want to restrict their use of an AF_INET6 socket to IPv6 communications only. For these applications the IPV6_V6ONLY socket option is defined. When this option is turned on, the socket can be used to send and receive IPv6 packets only. By default this option is turned off.

### Loopback address (::1 corresponding to 127.0.0.1 in IPv4)

Packets with the loopback address must never be sent on a link or forwarded by an IPv6 router. There are two separate loopback addresses for IPv4 and IPv6 and they are treated as such.

IPv4 and IPv6 addresses are separate address spaces except when it comes to "::".

### Compatibility address ::w.x.y.z

This is used for hosts and routers to dynamically tunnel IPv6 packets over IPv4 routing infrastructure. It is meaningful for OS kernel and routers. Java provides a utility method to test it.

### IPv4-mapped address ::ffff:w.x.y.z

This is an IPv6 address that is used to represent an IPv4 address. It allows the native program to use the same address data structure and also the same socket when communicating with both IPv4 and IPv6 nodes. Thus, on a dual-stack node with IPv4-mapped address support, an IPv6 application can talk to both IPv4 and IPv6 peer. The OS will do the underlying plumbing required to send or receive an IPv4 datagram and to hand it to an IPv6 destination socket, and it will synthesize an IPv4-mapped IPv6 address when needed.

For Java, it is used for internal representation; it has no functional role. Java will never return an IPv4-mapped address. It understands IPv4-mapped address syntax, both in byte array and text representation. However, it will be converted into an IPv4 address.

## IPv6-Related System Properties

On dual stack machines, system properties are provided for setting the preferred protocol stack—IPv4 or IPv6—as well as the preferred
address family types—inet4 or inet6.

IPv6 stack is preferred by default, since on a dual-stack machine IPv6 socket can talk to both IPv4 and IPv6 peers.

This setting can be changed through the `java.net.preferIPv4Stack=<true|false>` system property.

By default, we would prefer IPv4 addresses over IPv6 addresses, i.e., when querying the name service (e.g., DNS service), we would return Ipv4 addresses ahead of IPv6 addresses. There are two reasons for this choice:

1. There are some applications that expect an IPv4 address textual format, i.e. "%d.%d.%d.%d". Using an IPv4 address minimizes the surprises;

2. Using IPv4 address, we can use one call (with an IPv6 socket) to reach either a legacy IPv4-only service, or an IPv6 service (unless the IPv6 service is on a Ipv6 only node).

This setting can be changed through the system property `java.net.preferIPv6Addresses=<true|false>`

### Dual-Stack Node

For many years, if not forever, there will be a mix of IPv6 and IPv4 nodes on the Internet. Thus compatibility with the large installed base of IPv4 nodes is crucial for the success of the transition from IPv4 to IPv6. Dual stack, defined in RFC 1933, is one of the main mechanisms for guaranteeing a smooth transition. The other mechanism is IPv6 packet tunneling, which is relevant to the JDK only through the IPv4-compatible address. The former is the most relevant piece to the JDK. A dual stack includes implementations of both versions of the Internet Protocol, IPv4 and IPv6.

A general property of a dual-stack node is that an IPv6 socket can communicate both with an IPv4 and IPv6 peer at the transport layer (TCP or UDP) . At the native level, the IPv6 socket communicates with an IPv4 peer through an IPv4-mapped IPv6 address. However, unless a socket checks for the peers address type, it won't know whether it is talking to an IPv4 or an IPv6 peer. All the internal plumbing and conversion of address types is done by the dual-protocol stack.

**Note:** IPv4-mapped address has significance only at the implementation of a dual-protocol stack. It is used to *fake* (i.e., appear in the same format as) an Ipv6 address to be handed over to an IPv6 socket. At the conceptual level it has no role; its role is limited at the Java API level. Parsing of an IPv4-mapped address is supported, but an IPv4-mapped address is never returned.

### Java Application Impact

1. There should be no change in Java application code if everything has been done appropriately. I.e., there are no direct references to IPv4 literal addresses; instead, hostnames are used.
2. All the address or socket type information is encapsulated in the Java networking API.
3. Through setting system properties, address type and/or socket type preferences can be set.
4. For new applications Ipv6-specific new classes and APIs can be used.

**communication scenarios:**

| (Nodes) | V4 Only | V4/V6 | V6 Only |
|---|---|---|---|
| **V4 Only** | x | x | |
| **V4/V6** | x | x | x |
| **V6 Only** | | x | x |

Top row and left column represent various node types attempting to communicate. An x indicates that these nodes can communicate with each other.

**UDP**
**scenario 1:**

Either host1 or host2 can be a native application.

**host1 is server, host2 is client**

If host2 wants to talk to host1, it will create a V6 socket. It then looks up the IP address for host1. Since host1 only has a v4 protocol stack, it will only have IPv4 records in the name lookup service. So host2 will try to reach host1 using an IPv4-mapped address. An IPv4 packet will be sent by host2, and host1 will think it is communicating with a v4 client.

**host1 is client, host2 is server**

If host2 is the server, it will first create a v6-type socket (by default), then it will wait for connections. Since host1 supports v4 only, it creates a v4-type socket. They resolves the name for host2. It only gets v4 address for host2, since it doesn't understand IPv6 address. So it connects to host2 using v4 address. A v4 packet will be sent on the wire. On host2, the dual stack will convert the v4 packet into a v6 packet with a v4-mapped address in it and hand it over to the v6 socket. The server application will handle it as if it is a connection from a v6 node.

### Class Changes

**`InetAddress`**

This class represents an IP address. It provides address storage, name-address translation methods, address conversion methods, as

well as address testing methods. In J2SE 1.4, this class is extended to support both IPv4 and IPv6 addresses. Utility methods are added to check address types and scopes. The two types of addresses, IPv4 and IPv6, can be distinguished by the Java type `Inet4Address` and `Inet6Address`.

Two new subclasses of `InetAddress` are created: `Inet4Address` and `Inet6Address`. V4- and V6-specific state and behaviors are implemented in these two subclasses. Due to Java's object-oriented nature, an application normally only needs to deal with `InetAddress` class—through polymorphism it will get the correct behavior. Only when it needs to access protocol-family-specific behaviors, such as in calling an IPv6-only method, or when it cares to know the class types of the IP address, will it ever become aware of `Inet4Address` and `Inet6Address`.

The new methods introduced are:

```
InetAddress:
isAnyLocalAddress
isLoopbackAddress
isLinkLocalAddress
isSiteLocalAddress
isMCGlobal
isMCNodeLocal
isMCLinkLocal
isMCSiteLocal
isMCOrgLocal
getCanonicalHostName
getByAddr
```

**`Inet6Address:`**
**`isIPv4CompatibleAddress`**

#### `InetAddress` and Different Naming services

Prior to 1.4, `InetAddress` utilized the system configured name service to resolve host names. In 1.4, we have added a Java DNS provider through JNDI for alternative name lookups. You can tell the JDK to use this provider by setting up a few system properties. These system properties are documented in the Java system properties section. In the future, we plan to provide a generic service provider framework so that you can write your own name service providers.

#### A Word on Serialization

All IPv4 addresses are represented in Java as `Inet4Address` objects. They are serialized as `InetAddress` objects, and deserialized from `InetAddress` to `Inet4Address` to keep backward compatibility. IPv6 addresses are represented as `Inet6Address` and are serialized as such.

#### `Socket`, `ServerSocket`, and `DatagramSocket`

Due to the object-oriented nature of Java, the address types and storage structures are not exposed at the socket API level, so no new APIs are needed. The existing socket APIs handle both IPv4 and IPv6 traffic.

The selection of which stack to use depends upon the following:

1. The underlying OS support;
2. The user's stack preference property setting.

All supported Ipv6 socket options have a IPv4 counterparts. Thus no new APIs were added to support IPv6 socket options. Instead, the old APIs are overloaded to support both V4 and V6 socket options.

#### `MulticastSocket`

Again all the socket options APIs are overloaded to support IPv6 multicast socket options.

We have added two new APIs to set/get network interfaces in addition to the existing `MulticastSocket.setInterface`/`MulticastSocket.getInterface` that takes/returns an `InetAddress` instance. The two existing methods are used to set or retrieve the network interface used by the current `MulticastSocket` to send multicast packets (i.e., equivalent to `IP_MULTICAST_IF` in native socket option). For IPv4, the interface was indicated by an IP address. Thus we can use the equivalent `InetAddress` in Java. They will continue to work with IPv6 multicast socket. However, in IPv6, according to RFC 2553, the interface should be indicated using an interface index. To better support the concept of a network interface, we introduced a new class, `NetworkInterface`. It encapsulate the data representing the state of the network interface, including name and IP addresses and some basic manipulation methods. Thus we have introduced two new methods for setting the outgoing interface for multicast socket: `setNetworkInterface` and `getNetworkInterface`. They take or return a `NetworkInterface` object. These new methods can be used with both v4 and v6 multicast.

Methods have also been added for joining and leaving a multicast group on a network interface. This was previously unavailable in

the Java API.

```
MulticastSocket:
NetworkInterface getNetworkInterface()
setNetworkInterface(NetworkInterface netIf)
joinGroup(SocketAddress mcastaddr,NetworkInterface netIf)
leaveGroup(SocketAddress mcastaddr,NetworkInterface netIf)
```

#### `URL`, `URI` parsers

Literal IP addresses can be used in URL/URIs. However, since colon (`:`) is used in existing URL/URI specifications to separate host from port, using literal IPv6 representation in URL/URIs without modification will fail in parsing. Thus for specifying literal IPv6 addresses in URL/URIs, RFC 2732 was created. The parsing of URL/URI has been updated to be compliant with RFC 2732.

#### SocketPermission

Since `SocketPermission` utilizes URLs, its implementation has been updated to be compliant with RFC 2732.

`codebase`, used in defining a permission, is a variant of URL. As such, it should follow URL formats and conventions. RFC 2732 format is used for URL and `codebase`; RFC 2373 format is used everywhere else.

### IPv6 Networking Properties

#### `java.net.preferIPv4Stack (default: false)`

If IPv6 is available on the operating system, the underlying native socket will be an IPv6 socket. This allows Java(tm) applications to connect too, and accept connections from, both IPv4 andIPv6 hosts.

If an application has a preference to only use IPv4 sockets, then this property can be set to true. The implication is that the application will not be able to communicate with IPv6 hosts.

#### `java.net.preferIPv6Addresses (default: false)`

If IPv6 is available on the operating system, the default preference is to prefer an IPv4-mapped address over an IPv6 address. This is for backward compatibility reasons—for example, applications that depend on access to an IPv4-only service, or applications that depend on the %d.%d.%d.%d representation of an IP address.
This property can be set to try to change the preferences to use IPv6 addresses over IPv4 addresses. This allows applications to be tested and deployed in environments where the application is expected to connect to IPv6 services.

### JNDI DNS service provider settings:

#### `sun.net.spi.nameservice.provider.<n>=<default|dns,sun|...>`

Specifies the name service provider that you can use. By default, Java will use the system-configured, name-lookup mechanism, such as file, nis, etc. You can specify your own by setting this option. `<n>` takes the value of a positive number and it indicates the precedence order: a small number takes higher precendence over a bigger number. In 1.4 , we have provided one DNS name service provider through JNDI, which is called `dns,sun`.

#### `sun.net.spi.nameservice.nameservers=<server1_ipaddr,server2_ipaddr ...>`

You can specify a comma separated list of IP addresses that point to the DNS servers you want to use.

#### `sun.net.spi.nameservice.domain=<domainname>`

This property specifies the default DNS domain name, e.g., `eng.sun.com`.